PSO 2 — Week 5: Solutions

CS381, Fall 2026

Problem 1

(a) The table, a witness, and why three strings need a third index

With $L[i][0] = L[0][j] = 0$,

$$L[i][j] \;=\; \begin{cases} L[i-1][j-1] + 1, & A[i] = B[j], \\ \max\big(L[i-1][j],\, L[i][j-1]\big), & A[i] \neq B[j]. \end{cases}$$

Correctness. Induction on $i + j$. Fix $i, j \geq 1$, let C be a longest common subsequence of $A[1..i]$ and $B[1..j]$, and fix a realization of it: positions $p_1 < \cdots < p_t$ in A and $q_1 < \cdots < q_t$ in B with $A[p_r] = B[q_r] = C[r]$.

Case $A[i] \neq B[j]$. We claim C cannot use both position i of A and position j of B. Indeed i is the largest index available on the A side, so using it means $p_t = i$, and likewise using j means $q_t = j$; but p_t and q_t are matched to each other, which would give $A[i] = C[t] = B[j]$. So C skips position i or skips position j, making it a common subsequence of $A[1..i-1], B[1..j]$ or of $A[1..i], B[1..j-1]$, and $t \leq \max(L[i-1][j], L[i][j-1])$. Conversely both of those prefix pairs sit inside $A[1..i], B[1..j]$, so their common subsequences are common subsequences here too, and the reverse inequality holds.

Case $A[i] = B[j] = c$. Appending c to a longest common subsequence of $A[1..i-1]$ and $B[1..j-1]$ gives $L[i][j] \geq L[i-1][j-1] + 1$. The content of this case is the other direction, and for that it is enough to show C has a realization with $p_t = i$ and $q_t = j$: deleting its last character then leaves a common subsequence of the two shorter prefixes, so $t - 1 \leq L[i-1][j-1]$.

If $p_t < i$ and $q_t < j$, then positions i and j are both still free and we could append the pair (i, j), making $C \cdot c$ a longer common subsequence and contradicting optimality. So $p_t = i$ or $q_t = j$. If $p_t = i$, then $C[t] = A[i] = c = B[j]$, so we may re-match $C[t]$ to position j of B instead of q_t; the positions stay increasing because $q_{t-1} < q_t \leq j$. If instead $q_t = j$, re-match $C[t]$ to position i of A by the same argument. Either way we reach a realization with $p_t = i$ and $q_t = j$. $\square$

So checking that the pair (i, j) is one available option proves only $L[i][j] \geq L[i-1][j-1] + 1$; the exchange above is what rules out a longer common subsequence that avoids that pair.

Time and space. $\Theta(nm)$ entries at $O(1)$ each: $\Theta(nm)$ time and $\Theta(nm)$ space.

Recovering a witness. Walk backwards through the finished table from $(i, j) = (n, m)$: if $i = 0$ or $j = 0$, stop; if $A[i] = B[j]$, prepend $A[i]$ to the output and move to $(i-1, j-1)$; otherwise move to $(i-1, j)$ if $L[i-1][j] \geq L[i][j-1]$ and to $(i, j-1)$ if not. Each step decreases $i + j$ by at least one, so the walk takes at most $n + m$ steps.

The walk needs no extra bookkeeping, because at each cell the stored numbers already say which neighbour the value came from: on a match the recurrence gives $L[i][j] = L[i-1][j-1] + 1$, so the diagonal neighbour is the right one, and on a mismatch $L[i][j]$ is whichever of the two neighbours is larger.

The invariant is this: whatever the walk has emitted so far, prefixing it with any longest common subsequence of the current $A[1..i]$ and $B[1..j]$ yields a longest common subsequence of A and B. It

holds at the start, the two cases above are exactly the statement that each move preserves it, and at the end one of the prefixes is empty, so the emitted string is itself a longest common subsequence.

Three strings. Take

$$A = \texttt{aab}, \qquad B = \texttt{baa}, \qquad C = \texttt{b}.$$

Here A and B have `aa` as their *only* longest common subsequence, so the composition is well defined; it is not failing because of an arbitrary choice among ties. But `aa` and `b` have no character in common, so the composition reports 0, whereas `b` is a subsequence of all three strings and the true answer is 1.

The composition throws away every common subsequence of A and B but one, and that is fatal because the discarded ones are not recoverable from the survivor: `b` is a common subsequence of A and B that is not a subsequence of `aa`. A correct algorithm keeps the three prefixes as separate coordinates, using $L[i-1][j-1][k-1]+1$ when $A[i] = B[j] = C[k]$, and otherwise the largest of $L[i-1][j][k]$, $L[i][j-1][k]$ and $L[i][j][k-1]$. That costs $\Theta(nmp)$ time for strings of lengths n, m and p.

(b) Edit distance without substitutions

Write $L = L(A, B)$.

Upper bound. Fix a common subsequence C of length L, together with an occurrence of it in A and an occurrence in B. Delete the $n-L$ characters of A that the occurrence in A does not use; what remains is exactly C. Now insert the $m-L$ characters of B that the occurrence in B does not use, each at the position it holds in B; what remains is exactly B. That is $(n-L)+(m-L) = n+m-2L$ operations.

Lower bound. Consider any script of a deletions and b insertions turning A into B. Each deletion shortens the string by one and each insertion lengthens it by one, so $n - a + b = m$. Since there are no substitutions, a character of A that is never deleted is never altered, and insertions and deletions do not reorder surviving characters, so the $n - a$ undeleted characters of A appear in B in their original relative order. They therefore form a common subsequence, giving $n - a \le L$, that is $a \ge n - L$, and then $b = m - n + a \ge m - L$. Hence $a + b \ge n + m - 2L$. $\square$

Comparison with edit distance. Every insertion/deletion script is an edit script, so $d \le d_{\mathrm{id}}$. Conversely each substitution can be replaced by one deletion followed by one insertion, so $d_{\mathrm{id}} \le 2d$. The second inequality is tight, not just asymptotically: for $A = \texttt{a}^p$ and $B = \texttt{b}^p$ we have $L = 0$, so $d_{\mathrm{id}} = 2p$, while p substitutions give $d = p$. So the longest common subsequence cannot be used to read off the edit distance.

The two recurrences. On the prefixes $A[1..i]$, $B[1..j]$, with the obvious boundaries,

$$D_{\mathrm{id}}[i][j] = \min\big(D_{\mathrm{id}}[i-1][j]+1,\ D_{\mathrm{id}}[i][j-1]+1,\ D_{\mathrm{id}}[i-1][j-1] \text{ if } A[i] = B[j]\big),$$
$$D[i][j] = \min\big(D[i-1][j]+1,\ D[i][j-1]+1,\ D[i-1][j-1] + [\,A[i] \ne B[j]\,]\big).$$

The only difference is whether the diagonal move is available when $A[i] \ne B[j]$: forbidding it is exactly forbidding substitutions.

(c) When A has no repeated character

Because the characters of A are distinct, each character c of A has a unique index there; call it $\mathrm{pos}(c)$. Delete from B every character that does not occur in A at all, and let S be the sequence of

values $\mathrm{pos}(B[j])$ taken over the surviving positions j in increasing order. In other words, rewrite B as the list of places in A that its characters point to. Then $L(A, B)$ equals the length of the longest *strictly* increasing subsequence of S.

For instance, with $A = \texttt{cab}$ and $B = \texttt{bxca}$ we have $\mathrm{pos}(\texttt{c}) = 1$, $\mathrm{pos}(\texttt{a}) = 2$, $\mathrm{pos}(\texttt{b}) = 3$; dropping the `x` leaves `bca` and $S = 3, 1, 2$. Its longest strictly increasing subsequence is $1, 2$, of length 2, and indeed `ca` is a longest common subsequence of A and B.

The reason to want strictly increasing subsequences is that they encode exactly the two requirements on a common subsequence: reading S left to right respects the order of positions in B, and increasing values respect the order of positions in A.

From common subsequences to increasing subsequences. A common subsequence of length t is a pair of index sequences $p_1 < \cdots < p_t$ in A and $q_1 < \cdots < q_t$ in B with $A[p_r] = B[q_r]$. Each $B[q_r]$ occurs in A, so position q_r survives into S; and because the characters of A are distinct, p_r is the *only* index of that character, so $p_r = \mathrm{pos}(B[q_r])$. The entries of S coming from $q_1 < \cdots < q_t$ are therefore $p_1 < \cdots < p_t$: a strictly increasing subsequence of length t.

And back. A strictly increasing subsequence of S of length t comes from indices $q_1 < \cdots < q_t$ of B with $\mathrm{pos}(B[q_1]) < \cdots < \mathrm{pos}(B[q_t])$. Put $p_r = \mathrm{pos}(B[q_r])$. Then $p_1 < \cdots < p_t$ and $A[p_r] = B[q_r]$, so this is a common subsequence of length t. The two constructions are inverse to each other, so the maxima agree. □

Proving only the first direction would show that the longest increasing subsequence is at least $L(A, B)$, which is the wrong inequality for an algorithm that returns it.

Time. Building pos and scanning B takes $O(n + m)$ with a hash table, and patience sorting gives the longest increasing subsequence of S in $O(m \log m)$: total $O(n + m \log m)$. The hash table is where the algorithm assumes more than the ability to compare two characters; if comparisons are all that is available, sort the pairs $(A[i], i)$ and binary search each $B[j]$, for $O\big((n+m)\log n\big)$ overall — at $n = m$, $O(n \log n)$ against the table's $\Theta(n^2)$.

Repeated characters. With repeats there is no pos to speak of: a character of B may match several positions of A, so the reduction has no single value to write down for it. The standard repair writes down all of them — replace each surviving $B[j]$ by the list of every position of $B[j]$ in A, in *decreasing* order, and concatenate the lists. A strictly increasing subsequence still corresponds to a common subsequence, and the decreasing order inside a block is what stops two positions of the same block from being used together. But the concatenated sequence has length $r = \#\{(i, j) : A[i] = B[j]\}$, and $r = nm$ already for $A = B = \texttt{a}^n$, so the resulting $O\big((r + m)\log n\big)$ bound is quadratic in the worst case. That is not an accident of the method: no algorithm running in $O(n^{2-\epsilon})$ for a fixed $\epsilon > 0$ is known, and none exists under the strong exponential time hypothesis.